FaaS for IoT: Evolving Serverless towards Deviceless in I/Oclouds

Giovanni Merlino, Giuseppe Tricomi, Luca D'Agati, Zakaria Benomar, Francesco Longo, Antonio Puliafito

Future Generation Computer Systems 154 (2024) 189–205

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

FaaS for IoT: Evolving Serverless towards Deviceless in I/Oclouds

Giovanni Merlino [a,b], Giuseppe Tricomi [a,b,d,*], Luca D'Agati [a], Zakaria Benomar [c], Francesco Longo [a,b], Antonio Puliafito [a,b]

[a] University of Messina, Italy
[b] CINI: National Interuniversity Consortium for Informatics, Rome, Italy
[c] Inria: Institut National de Recherche en sciences et technologies du numérique, Paris, Île-de-France, France
[d] ICAR-CNR: Institute of High-Performance Computing and Networking (ICAR) of National Research Council of Italy (CNR), Napoli, Italy

ARTICLE INFO

Keywords:
IoT
Serverless
Deviceless
FaaS
Edge computing
Stack4Things

ABSTRACT

The burgeoning paradigms of Fog and Edge computing propose delegating Cloud-related tasks to the network's periphery, thus placing computational resources closer to data producers. This shift promises to boost the performance of IoT-based services, providing swift response times while conserving bandwidth. Despite their potential, the current Edge/Fog computing platforms must provide the required flexibility for dynamic service orchestration within a data-oriented context. Addressing this gap, the Function-as-a-Service (FaaS) model emerges as an exceptional strategy for Edge/Fog deployments. Its ability to manage an ever-expanding ecosystem of devices with remarkable flexibility and efficiency holds considerable promise. This paper articulates a novel approach to enhancing the adaptability of IoT Edge/Fog deployments. We propose an innovative extension to OpenStack, an open-source Cloud management system, which pushes its functionality towards the network Edge. Our approach empowers OpenStack to facilitate FaaS services within a distributed IoT infrastructure, thus infusing unprecedented adaptability and efficiency into the Edge/Fog computing paradigms.

1. Introduction

The information technology (IT) domain has experienced unprecedented expansion in recent years, catalyzed by increasing competitive pressure among service providers to accelerate their value generation and service deployment. The emphasis on reducing time-to-market, traditionally plagued by lengthy infrastructural procurement, configuration, deployment, and manual software management tasks, is now more significant than ever. The widespread practice of server overprovisioning for improved scalability and resilience has often led to the underutilization of hardware resources, yielding inefficiencies in terms of time and cost.

The advent of virtualization technologies over the past decade has revolutionized the IT landscape, fostering optimized usage of hardware resources and enabling cost-effective operations [1,2]. Consequently, this has led to the emergence of the Cloud computing paradigm, wherein hardware resources are treated as on-demand utilities. In this paradigm, companies need not hold physical hardware, like servers, but can access and leverage remote infrastructure via the Internet, only paying for the resources consumed.

For a considerable period, "virtualization" primarily referred to Hypervisor-based virtualization. However, advancements in the field have given rise to an alternative form of virtualization, Operating System (OS)-level virtualization, better known as containerization, which has gained popularity due to its more efficient resource utilization, faster provisioning/de-provisioning, and superior scalability [3].

Although Hypervisor-based virtualization continues to provide users with on-demand access to computing resources, containerization adds facilities for self-contained applications on demand. This shift has catalyzed an evolution from large monolithic applications to distributed architectures based on microservices [4]. Containers and microservice-based architectures are paving the way for a new paradigm, Serverless computing, oftentimes also known as, albeit not the same concept as, Function-as-a-Service (FaaS) [5].

Despite its name, the serverless model involves infrastructure, namely servers, managed entirely by the service providers, rendering them transparent to developers and end-users. This model enables developers to focus primarily on the business logic and deployment of features through functions without worrying about infrastructure management and runtimes' configurations. Functions being stateless and independent enhance systems scalability and resilience overall, leading to a revolution in software engineering procedures for modern software

* Corresponding author at: University of Messina, Italy.
E-mail address: gtricomi@unime.it (G. Tricomi).

https://doi.org/10.1016/j.future.2023.12.029
Received 9 September 2023; Received in revised form 24 November 2023; Accepted 28 December 2023
Available online 2 January 2024

delivery [5]. Simultaneously, the explosive growth of the Internet of Things (IoT) has catalyzed an evolution in smart environments and Cyber–Physical Systems (CPS), where devices are no longer mere data sources but participants in complex data processing pipelines. While providing comprehensive management facilities, Cloud computing has shown limitations when dealing with the demands of real-time, localized IoT services. Consequently, paradigms such as Fog and Edge computing have emerged, bringing (e.g., computational) resources and/or tasks closer to data producers and consumers [6].

The FaaS model, coupled with these Edge/Fog computing approaches, proves beneficial for IoT applications by deploying personalized functions onto IoT devices for localized and efficient responses to events [7]. IoT services, in this context, evolve from merely collecting raw data to producing actionable results, improving user Quality of Experience (QoE) by reducing latency, and enhancing security by isolation of execution units and/or runtimes [7,8].

In this paper, our contribution is threefold. We propose an OpenStack-based framework for FaaS, capitalizing on IoT resources for function hosting; we introduce a preliminary integration of the OpenStack subsystems for, respectively, containers, functions, and IoT, Zun, Qinling, and IoTronic; and we provide an evaluation that underlines the advantages of our solution in the context of the I/Ocloud paradigm, concerning application definition, configuration, and resource usage at the Edge devices. Furthermore, we propose a solution to manage the FaaS approach for the I/Ocloud component mentioned above through a unified system relying on Node-RED. The novelties and contribution presented in this work may be summarized as follow: (i) IoT fleet management as the only prerequisite to swiftly deploy FaaS-based applications on demand and at will on top of managed nodes, (ii) a tool able to define complex applications as workflows composed of blocks. The workflow definition and deployment are enabled by leveraging the FaaS paradigm. (iii) simplified functions configuration without direct device interaction mediated via a user-friendly dashboard, (iv) IoT device facilities may be used for heterogeneous purposes through the injection of additional functions on devices originally assigned to other workflows, (v) extensibility and scalability of applications improved through FaaS-enhanced infrastructure management.

The remainder of this paper is structured as follows: Section 2 presents an analysis of literature concerning the scenario in which the proposed solution works; Section 3 introduces key concepts and building blocks. Section 4 provides a high-level overview of the architecture, while Section 5 delves into the main mechanisms and workflows of the solution. Section 6 describes a use case leveraged to analyze the framework under consideration. The experimental results are discussed and presented in Section 7 to complete the evaluation. Lastly, Section 8 discusses the results presented in this paper and outlines future directions for this research activity.

2. Related work

2.1. Cloud and Fog in IoT

In recent years, significant efforts have been spent to promote the Cloud as a suitable paradigm for managing IoT environments. Indeed, several methods and techniques have been introduced to deal with the management of remote and resource-constrained infrastructure. In this context, a set of issues has been addressed in the literature, such as scalability, device accessibility, and personalization of services. To have extensive insight into the challenges in integrating the Cloud and IoT, readers may refer to [9,10]. In the same perspective, several platforms were introduced to aggregate IoT deployments under the scope of Cloud management.

Despite the wide range of benefits the Cloud paradigm provides (e.g., pooling of storage and computing resources), novel constraints in terms of Quality of Service (QoS) dictated by current and forthcoming applications make the Cloud unfit to meet the corresponding requirements. The Fog/Edge computing paradigms push the resources close to the data producers (e.g., storage and computing) to the network edge, reaching the goal of overcoming the intrinsic limitations of the Cloud paradigm stated above. Fog computing nodes are bound to be close to data sources, which is a key enabler of advanced applications [6] that were not feasible when relying only on the faraway Cloud infrastructure.

2.2. Serverless vs. Function-as-a-service (FaaS)

Both approaches are related to the trend to virtualize resources and provide these as utility, introduced by Cloud Computing. However, we can move on one or the other according to how virtualization activities are managed. Until 2017, authors confused these two paradigms, referring to them without discriminating whether a full-stack environment or the execution of a simple function is required, as in [5,11], and a few other works. In particular, in [12], they define serverless as: *"a software architecture where an application is decomposed into 'triggers' (events) and 'actions' (functions), and there is a platform that provides a seamless hosting and execution environment."* that correspond more or less to the merge of the definitions used for serverless and FaaS. On the Web, it is possible to identify several works discussing the differences between the two approaches and even respective advantages/disadvantages, as in [13] and, depending on the aspects under analysis, some [14] commend the serverless approach, and others [15] highlight the FaaS approach as one of the main trends currently. Identifying the traits characterizing these two concepts is very important to understand how the workflow is managed in the presented architecture and why some architectural choices have been made. Recently, the Cloud Native Computing Foundation (CNCF) [16] disambiguated concepts related to serverless computing by introducing the Backend-as-a-Service (BaaS) terminology in the process. They state that, by definition: (i) **Function-as-a-Service** provides small units of code, representing event-driven computing facilities, where the functions get instantiated and triggered from an external source, typically through commonplace HTTP requests; (ii) **Backend-as-a-Service** is an approach to have common backend tasks be handled without any customer's involvement in their management; (iii) **Serverless** is defined as the combination of FaaS and BaaS.

2.3. Fog/Edge computing and Serverless/FaaS

With the evolution of Cloud solutions, all major Cloud service providers nowadays include, among their offerings, Serverless computing solutions. For instance, Amazon Web Services (AWS) has AWS Lambda,[1] enabling consumers to run their code without provisioning the infrastructure. IBM also provides a Serverless platform named IBM Cloud Functions,[2] built on top of Apache OpenWhisk [17]. The same is true for Microsoft Azure and Google, which propose Cloud-backed Serverless plans in the form of Azure Functions[3] and Google Functions,[4] respectively; thus, their consumers can deploy their functions in the Cloud. With the proliferation of IoT devices, the amount of data generated at the network Edge has experienced immense growth. Examples include sensor data, events generated by IoT devices and gateways, and multimedia files such as camera images. To use this data and provide new services/applications with added value, the incumbent cloud players show immense interest in the Fog/Edge paradigms and promote the Serverless/FaaS approaches as suited solutions to be adopted at the network edge. In fact, Microsoft has released a

[1] https://aws.amazon.com/lambda/
[2] https://cloud.ibm.com/functions/
[3] https://azure.microsoft.com/en-us/services/functions/
[4] https://firebase.google.com/

Fog/Edge platform for IoT called Microsoft Azure IoT Edge [18] that extends the Cloud Serverless paradigm towards the network Edge using containerization technology. Likewise, Amazon and IBM extended their pre-existing proprietary Cloud solutions to the network Edge using AWS Greengrass [19] and IBM Watson IoT [20] platforms, respectively. AWS Greengrass, for instance, allows users to run AWS Lambda functions on Edge devices; hence, they can deploy customized applications on IoT devices. Although opting for Serverless offerings from public Cloud service providers is a widely adopted strategy to deploy applications, one of the most significant issues related to public Cloud Serverless solutions is definitely vendor lock-in. Indeed, Cloud providers can impose their own choices for strongly (user-)restrictive configuration settings, e.g., caps for the execution duration of functions or concurrent executions.

Moreover, in this setting, data privacy, sovereignty, and control of owned infrastructure are relinquished and cannot be easily reclaimed by the IoT owner and/or IoT-hosted service user. Such concerns can be addressed and solved if the Serverless paradigm is deployed using a private Cloud environment. Within this context, our Stack4Things (S4T) middleware provides an open-source solution based on industry-standard protocols and services that can run on-premises without relying on third-party data centers. An administrator can deploy his/her own self-controlled private Cloud; thus, he/she can have total control over the deployment settings and configurations. In the literature, as reported in [21], several works target the use of the Serverless/FaaS paradigms in IoT deployments, which are cataloged in a taxonomy that considers challenges, protocols, technologies, architectural insights, and use cases. The great variety of applications and use cases reported highlights the flexibility of the paradigm applied on Fog or Edge devices. For instance, authors in [22] proposed a Fog-based Serverless system that supports data-centric IoT services. In particular, they focused their work on a smart parking use case. In the same context, authors in [23] introduced a platform named Kappa that can be used to deploy functions on devices at the network Edge.

2.4. Comparing traditional FaaS with our deviceless solution

Recent IoT and cloud computing advancements have led to significant developments in FaaS solutions, each offering distinctive perspectives and methodologies. Studies such as those by Garbugli et al. [24] have delved into QoS management in distributed cloud environments, emphasizing the increasingly pivotal role of edge resources in serverless paradigms. This aligns with our work's objective of enhancing serverless computing in IoT through OpenStack integration, proposing a distinct approach in terms of implementation and architectural focus. Complementing this perspective, the research by Benedetti et al. [25] examines the application of serverless computing to IoT platforms, particularly addressing challenges in warm start and cold start deployment models within resource-constrained environments. This analysis significantly enriches our understanding of serverless computing performance, a critical aspect of integrating FaaS services within distributed IoT architectures. Despite these advancements, existing FaaS solutions often need to be revised, especially in addressing the complex requirements of IoT environments. These include resource management challenges, data processing latency, and difficulties in ensuring efficient deployment and scalability in highly distributed systems. Current FaaS models also sometimes grapple with the complexity of integrating diverse IoT platforms and maintaining consistent performance across varied devices and network conditions. Our approach seeks to address these challenges, offering an OpenStack-based framework that capitalizes on the strengths of cloud computing while optimizing for the distinct needs of IoT systems. By integrating serverless computing capabilities into OpenStack, we aim to enhance the adaptability and efficiency of distributed IoT architectures. We focus on both the performance and scalability aspects and infrastructure management of serverless functions within the IoT ecosystem, addressing immediate deployment needs and anticipating future scalability and infrastructure management challenges. Furthermore, innovations such as CSPOT and tinyFaaS, highlighted in [26,27], show the evolving landscape of FaaS solutions in IoT. CSPOT extends the FaaS model across IoT scales, focusing on application robustness and reducing latency. At the same time, tinyFaaS provides a tailored solution for edge environments, emphasizing resource efficiency and effective communication with low-power devices. These advancements demonstrate diverse approaches to scalability, interoperability, and performance optimization, offering perspectives that both contrast with and complement our approach.

3. Background

In this section, we explore the key technologies we used to conceive our Edge-based FaaS system, focusing on containerization as an enabler of the FaaS model within the cloud computing paradigm. We contrast the traditional Kubernetes-based container orchestration with our implementation using OpenStack's Zun and Qinling services. This comparison highlights how our approach, complemented by our Stack4Things (S4T) middleware, diverges from the conventional Kubernetes-dominated landscape in managing containerized environments, particularly within IoT contexts. Container-based virtualization, leveraging tools such as Namespaces, Control Groups (Cgroups), and Secure Computing Mode (Seccomp), provides a flexible and efficient alternative to Hypervisor-based virtualization. Our discussion extends to Kubernetes' architecture and its role in container orchestration, setting the stage to contrast it with OpenStack's Zun for container management and Qinling as an OpenStack FaaS service. The integration of these OpenStack components within our system represents a distinct approach, targeting enhanced adaptability and efficiency in distributed IoT architectures, a deviation from typical Kubernetes applications.

3.1. Container-based virtualization

The IT field has made significant progress with the set of benefits OS-level virtualization introduced [28]. This key open-source technology consolidates the virtualization realm with an efficient, lightweight alternative for Hypervisor-based virtualization. Actually, the image-based containerization approach gives considerable flexibility, thereby making it a Swiss army knife solution to be adopted in different environments and circumstances [29]. In the following, we provide a technical description of this technology based on a Linux environment as it fully supports containerization. Containers are provisioned in a Linux-based OS using a set of features provided by the Kernel, such as Namespaces for process isolation, Control Groups (Cgroups) for resource management, and Secure computing mode (Seccomp) for secure sandboxing. More in detail, the Namespacing feature [30] is responsible for process isolation. When a container gets created, the Kernel isolates its processes using Namespaces that create abstracted instances of particular system resources (e.g., Mount, UTS, IPC, PID, Network). Hence, a container ends up being a one-off instance with reference to the processes running inside it.

Consequently, multiple containers can run simultaneously on top of a shared OS without begetting any conflict or unauthorized visibility among them. Regarding the Cgroups capability [31], it manages the containers' allocation/control of the resources (e.g., RAM, CPU, block I/O, devices, network bandwidth). In particular, Cgroups allow a dynamic aspect while managing the allocation of resources, thereby making the containers flexible. Last but not least, Seccomp [32] is a feature the Linux Kernel provides that restricts the system calls that a process can use. For instance, when a developer uses potentially unsafe/unverified code or software, Seccomp efficiently isolates and restricts the code/program from using calls that have not been (already) permitted and declared. Thereby, the Seccomp capability prevents the misbehavior of launched applications inside containers that could gain access to the host Kernel and compromise it.

3.2. Kubernetes

Among the open-source management systems for containerized applications able to automate deployment and scaling, Kubernetes (K8s) is one of the most famous. This platform orchestrates tasks such as scaling containers up or down, deployment patterns, and distributing workloads among containers. Furthermore, by design, it follows the traditional Client-Server paradigm with a Master (node) responsible for decision-making, allocating resources, and scheduling containers; in a few words, it is designed to coordinate the deployment (i.e., cluster). Some of the essential elements of the Kubernetes ecosystem are:

- ***Clusters:*** a Kubernetes *cluster* is formed by several Nodes to pool their resources. These resources are then shared to be used in an abstract fashion by the applications deployed within the *cluster*.
- ***Nodes:*** A Node is a worker machine (either VMs or physical) belonging to a *cluster* managed by Kubernetes Master Node.
- ***Pods:*** a *Pod* is the basic building block in Kubernetes-based deployments. A *Pod* can host one or multiple containers depending on the application's needs. Containers inside the same *Pod* share their resources (e.g., network and storage) and can operate closely with each other to achieve a goal.
- ***Persistent Volumes:*** The containers use this useful mechanism to take advantage of permanent storage. The reason supporting its utilization is obvious if a reader considers that Kubernetes provides automatic up/down scaling containers that can be short-living instances as they get created/deleted dynamically; hence, each of their local data storage can be volatile.

3.3. OpenStack Containers service: Zun

Zun is one of the OpenStack projects aiming to manage container deployments. Zun's users may instantiate and use containers rapidly and without the need to manage their placement. This is possible because, at the backend, Zun can use a set of technologies to manage containers in a transparent/abstract manner, hence hiding the complexity of the workflows. One of the most used Zun configurations is based on the exploitation of Docker as a container runtime tool and (optionally) collaborating with other OpenStack subsystems such as Neutron to provide advanced networking capabilities (e.g., Load balancing, security groups) to the containers, and Glance to manage containers' images (instead of pulling them from public repositories).

The architecture of Zun consists of five main components:

- ***Zun-API:*** a WSGI server able to handle the users' REST requests from the CLI or Horizon dashboard.
- ***Zun-compute:*** is an agent that stands on the Compute nodes (where containers get instantiated). By performing (almost) all backend operations, this agent hides the workflows of the services.
- ***Zun-wsproxy:*** a WS-proxy server for providing container interactive mode and streaming.
- ***Zun scheduler:*** is a component responsible for decisions-making. In particular, when it receives a request to create a container, it uses a set of filters (e.g., RAM, CPU, etc.), and it interacts with the Container Orchestrator Engine (COE) (e.g., Kubernetes) to select the 'best' host where to deploy the container.
- ***Zun networking driver:*** is responsible for ensuring the network reachability for the containers. In Cloud-only OpenStack deployments, Zun delegates this task to a networking driver that interacts with the OpenStack networking subsystem, Neutron, that assigns the containers private IP addresses.

Another OpenStack subsystem that is an excellent candidate for exploiting the Zun COE facilities is the OpenStack FaaS project: Qinling. This way, we integrated the Zun orchestrator with Qinling to come up with our Edge-based FaaS system. This choice was made for a purpose, as OpenStack integration with Kubernetes is not totally straightforward and still needs some manual configurations. This way, to have a full and tight integration with OpenStack, using a built-in Zun orchestrator compatible with the other subsystems (i.e., Qinling) is a fit solution. Considering the similarity in basic concepts between Kubernetes and Zun, it is possible to associate the Kubernetes *Pod* with the Zun *Capsule*, a group of containers sharing the same network configuration, namespaced PID, Mount, etc. For *Persistent Volumes*, Zun can be extended to use different storage backend technologies such as Docker local volumes, Swift and Amazon S3. For our system, we opted for an integration with the Local Volumes provided by Docker.

3.4. OpenStack FaaS service: Qinling

Qinling is the OpenStack project to provide FaaS services, enabling its users to deploy functions based on the Serverless paradigm (like AWS Lambda). By design, Qinling may support several container orchestration platforms such as Kubernetes and Docker Swarm to manage and maintain the underlying *Pods* hosting the functions. For our system and to keep a full integration with OpenStack, our built-in Zun-based orchestrator will manage the *capsules* required by Qinling. The architectural design of Qinling consists of three main components:

- ***Qinling-API:*** is a WSGI server that handles the different requests received. For instance, it interacts with Keystone for authentication and then routes the requests to the Qinling-engine or the Qinling-orchestrator.
- ***Qinling-engine:*** is the central component that processes all the backend operations, such as runtime maintenance and function execution operations. Also, the engine maintains the mapping between the functions and the containers where they are deployed, as it has access to the database.
- ***Qinling-orchestrator:*** is the component that orchestrates/manages the FaaS deployments while interacting with a container orchestrator (e.g., Kubernetes or Zun). In particular, It manages the scaling up/down of containers depending on the charge of requests received by the system.

When a user requests the instantiation of a function, and before it gets executed, Qinling handles the event by preparing the environment needed to execute the function (e.g., packages needed, runtime). The Qinling request to create such an environment is translated into a Zun *capsule* generation. The *capsule* that gets created is composed of 3 containers:

- ***Runtime container:*** it is where a function code gets executed. Qinling supports three runtime environments (i.e., Python2, Python3, and Node.js)
- ***Sidecar container:*** it is responsible for downloading, from the Cloud, the functions' needed packages. The runtime container afterward uses these packages.
- ***Pause container:*** it represents the *capsule* and provides network capabilities to the other two containers since it is the only one attached to the network.

3.5. Stack4Things-based I/Ocloud

Stack4Things (S4T) is a research project aimed at expanding the universally adopted open-source Cloud management system, OpenStack, to accommodate Internet of Things (IoT) management as well [33]. This endeavor involves implementing various features and capabilities within the S4T middleware, enabling IoT deployments to participate in an Edge-extended Infrastructure as a Service (IaaS)/Platform as a Service (PaaS) Cloud. The principal vision of the S4T project is to facilitate users in leveraging IoT devices and their I/O resources, such as sensors and actuators, through well-defined APIs akin to using standard Cloud resources [34]. This leads to the emergence of a new

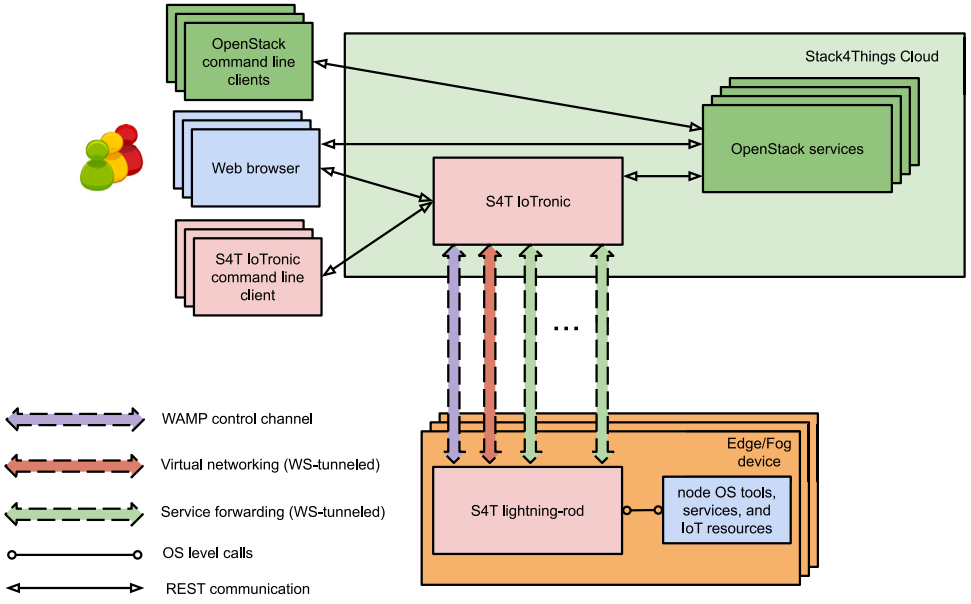

Fig. 1. Stack4Things core subsystems.

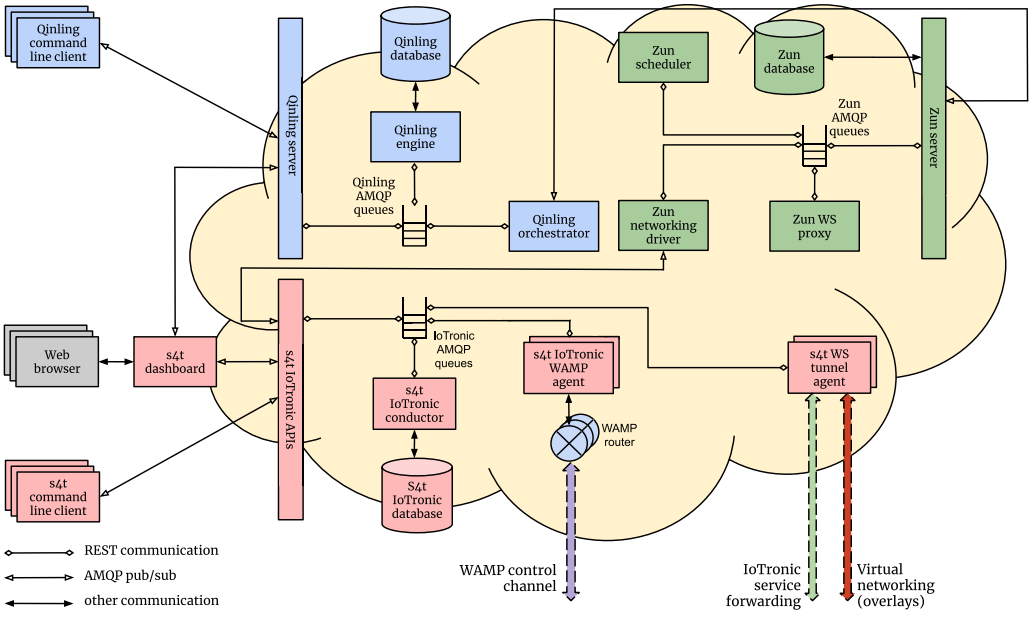

Fig. 2. Stack4Things FaaS Cloud-side subsystem design.

paradigm, I/Ocloud [35], which introduces IoT virtualization capabilities in tandem with conventional IaaS virtualization, like computing and storage.

In the I/Ocloud concept, Virtual Nodes (VNs) that can host the business logic and utilize the attached I/O resources are introduced, much like a real IoT device. These VNs can be deployed either at the datacenter-level infrastructure or directly on the physical IoT nodes at the Edge, depending on the service context, such as time sensitivity, required computational resources, and more. To cope with this objective, the FaaS paradigm represents the optimal candidate for IoT virtualization inside VNs. In this direction, the studies made in [36,37] pave the way for realizing a server/device-less approach suitable for I/Ocloud needs. The design of S4T deployment is divided between a Cloud data center and a collection of IoT devices, illustrated in Fig. 1. For the hardware configuration of the IoT nodes, single-board computers like Arduino and Raspberry Pi are utilized, which can host a minimal Linux distribution, such as OpenWRT. As a result, these devices can accommodate various Linux-based tools and different runtime environments like Python and Node.js, which are essential for the node-side S4T agent known as Lightning-Rod (LR) [38,39].

The S4T Cloud-side subsystem, IoTronic, is developed considering the standard architecture of OpenStack services, demonstrated in Fig. 2. This design ensures full compatibility with other OpenStack subsystems like Keystone, Neutron, and Qinling, among others. On the node side, the LR agent is a crucial component that links the IoT devices, even those deployed behind Network Address Translations (NATs) or strict firewalls, to the Cloud infrastructure where the S4T IoTronic service is deployed. The interconnections between the Cloud and the devices leverage the exploitation of WebSocket (WS) solutions to bypass networking middleboxes like NATs and firewalls. Notably, S4T utilizes the Web Application Messaging Protocol (WAMP), a subprotocol of WS, to create a full-duplex messaging channel for routing traffic streams,

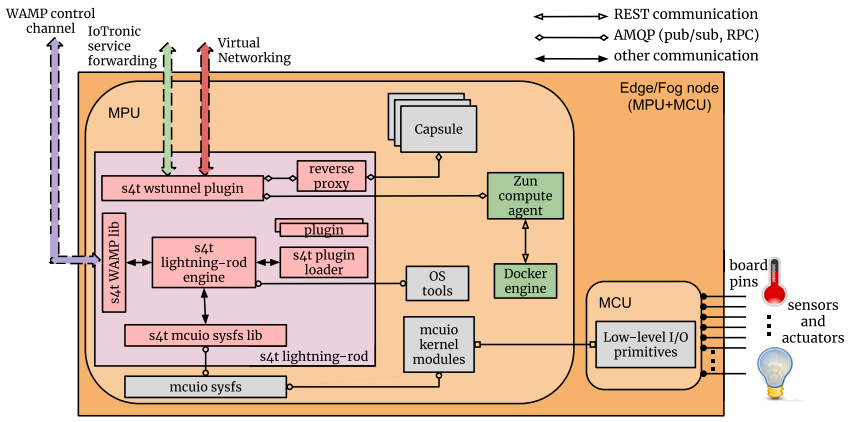

Fig. 3. Stack4Things FaaS Edge/Fog-side subsystem design.

such as commands, between the Cloud and the distributed IoT devices. WAMP offers two significant features: publish/subscribe (pub/sub) messages and Remote Procedure Calls (RPCs). The S4T middleware enables users to expose internal services (e.g., SSH, VNC) of an IoT device through the Cloud by deploying WS tunnels using a reverse mechanism known as 'rtunnel'.[5] The IoT devices initiate communication with the Cloud, as shown by the green arrows in Figs. 1, 2, and 3.

Moreover, a service request that reaches the Cloud on a specific port is forwarded to the S4T IoTronic WS tunnel agent, which controls the WS server to which a device connects through the wstunnel libraries (Fig. 3). The same 'rtunnel' mechanism creates overlays between distributed IoT devices, enabling them to reach each other and the Cloud-based instances (i.e., Virtual Machines, containers) as if they were on the same Local Area Network (LAN). The LR agent establishes connections between the Cloud and the devices and oversees all operations to be executed on the IoT devices, including management tasks or user needs, like interacting with the sensors and actuators. This unique approach significantly extends the potential for customization and reprogramming of IT infrastructures in various environments, such as smart buildings [40] and industrial settings [41], by fostering cooperation among different CPSs within a given space.

4. System architecture

This section presents an extension of the OpenStack Cloud-based management system to the network Edge. In particular, the approach uses our subsystem, IoTronic, and modified Qinling and Zun subsystems to make the OpenStack system able to provide FaaS services on top of distributed IoT devices located at the Edge of the network. This newly introduced functionality in OpenStack is achieved through RESTful interactions with Qinling that uses, in the backend, Zun and IoTronic to deploy functions on remote IoT devices. A significant difference from data center-based OpenStack deployments is that, in our approach, the three components, Zun-compute, Docker engine, and the runtimes, are not deployed on the Cloud. In fact, to extend the (limited) FaaS scope features provided by Qinling and Zun in standard Cloud deployments and make them able to manage FaaS services at the network Edge, we adapted the typical Cloud architecture design by deploying on the cloud side: the Qinling subsystem (engine, orchestrator, and API server), Zun (API server and Zun scheduler) and IoTronic, while on the IoT devices the other components: Zun-computes, Docker engines, and the runtimes that commonly are deployed on the Cloud as well when considering standard OpenStack deployment. Fig. 4 depicts the architecture design of our Edge-based FaaS system. In particular, we have a Cloud part that manages/coordinates the whole deployment and an Edge/Fog layer composed of the IoT devices. The users can interact with the management system through several methods (e.g., Dashboard, APIs, etc.). We organized the system's structure into three parts based on this architecture.

- ***User Interface (UI):*** it equips users with an easy way to interact with the system. In particular, the UI exploits the APIs exposed to provide a graphical interface to set deploy/manage functions on the IoT devices. We mention here that our system provides two UIs. The first one is dedicated for administration purposes and is based on the OpenStack standard dashboard, Horizon. The second one is intended to be used for applications/services purposes. It is designed using the Node-RED tool to allow end-users to use the system efficiently.
- ***Management Nodes:*** these components represent the Cloud-side of the system. This part of the system is responsible for running the services and provides suitable mechanisms to make the IoT devices reachable by the users. In our Edge-based FaaS system, the Cloud hosts Qinling, Zun, IoTronic, and Keystone. Still, it can be extended to host other subsystems, such as Neutron and Glance, to mention a few, to provide other advanced S4T services (e.g., virtual networking).
- ***Edge workers/Compute nodes:*** are the IoT devices deployed at the network Edge that hosts the LR agents. They can interact with the Cloud part using WS-based communication channels. In the I/Ocloud perspective, they represent the physical element in which the VNs functions run.

4.1. User Interfaces

The end-user UI based on Node-RED we provide with our system allows users to easily deploy their desired business logic and workflows on the IoT devices by instantiating functions managed by our FaaS system. In this way, the users can deploy on the Fog/Edge devices particular data workflow pipelines and software-defined actions using the resources they can host (e.g., sensors and actuators). Regarding the management/administration, the OpenStack dashboard (i.e., Horizon), which aims only at managing the infrastructure without dealing with users' needs, has also been extended to be aligned with the new features made available through the Edge-based FaaS system.

4.2. Management nodes

Our system's cloud side comprises several OpenStack services (e.g., Keystone, Neutron, Glance). We report in the following the three subsystems involved in our FaaS system (see Figs. 2 and 4).

[5] https://github.com/MDSLab/wstun

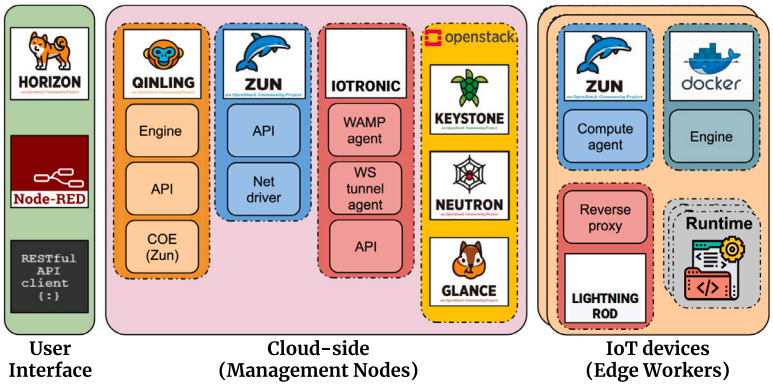

Fig. 4. The Edge-based Faas system architecture.

- ***Qinling:*** It is the subsystem responsible for managing the FaaS capabilities. In particular, it receives the users' requests and deals with function instantiation (see Section 3.4). In our Edge-based FaaS system, Qinling uses the Zun-based orchestrator that we integrated with the system instead of Kubernetes or Docker Swarm.
- ***Zun:*** it manages the life cycle of the containers required by the Qinling subsystem (see Section 3.3). Regarding the containers/*capsules* networking, Zun, in our use case, has been extended by introducing a new networking driver that uses IoTronic.
- ***IoTronic:*** it manages the life-cycle of the IoT devices and links them to the Cloud infrastructure. Besides, it can expose the devices' internal services and the containers they can host, using a Cloud public IP address and associated ports. This way, the internal services/containers become reachable even when the IoT devices are deployed behind some middleboxes (see Section 3.5). In particular, this capability is used in our FaaS system to enable users to reach their Edge-based runtimes. This service is integrated with Zun as a networking driver.

4.3. Edge workers

The last part of the system is made up of the IoT devices that host the LR agents, as depicted in Fig. 4. An IoT device, when belonging to the FaaS "pool" hosts the following components (see Figs. 3 and 4).

- ***LR agent:*** S4T-specific agent hosted on IoT devices. Its main duties consist of registering the devices to the Cloud and managing the commands/requests from users/administrators (see Section 3.5).
- ***Zun-Compute:*** This agent performs (almost) all backend operations related to the container life-cycle invoked by the Zun API server (see Section 3.3).
- ***Docker engine:*** It is responsible for creating and running Docker containers. It is based on a client–server architecture that uses HTTP. This engine receives management requests from the Zun-Compute agent in our use case.
- ***Reverse Proxy:*** This component is part of LR. It has an important role in the Edge/Fog FaaS system as it is responsible for forwarding the runtimes and function execution requests. In detail, the reverse proxy associates the *runtime_id* field contained in the requests received and the runtimes deployed on the device.

4.4. Bird's eye view of function and runtime deployment workflows

The system's architecture is highlighted in Figs. 2 and 4. To deploy a runtime/function on a particular IoT device, a user interacts, through the dashboard or CLI, with the Qinling-API server that forwards the request to the Qinling orchestrator. This latter component cooperates with the Zun-scheduler to identify the IoT device where the runtime/function should be deployed; then, the Zun-API server sends a request to create, on this device, the containers needed (i.e., the capsule). To enable users to reach the capsule, particularly the runtime container, IoTronic exposes it on the Cloud side, using a public IP address and a port; then, a WS tunnel is created between the Cloud and the IoT device. Hence, a request that reaches the Cloud on that IP address/port will be forwarded to the WS tunnel and reaches the device. On the device side (see Fig. 3), the request is received through the S4T wstunnel plugin and forwarded to the reverse proxy that routes it to the correct runtime.

4.5. Advantages of functions deployment at the network edge

The framework to use the FaaS approach for I/Ocloud allows users to deploy business logic in applications composed of multiple functions connected and running at the Edge/Fog levels. This approach can significantly improve the application setup, management, and upgrade. From the I/Ocloud perspective, the Virtual Nodes work side-by-side with the Node-RED dashboard, invoking functionalities executed as functions through the framework. The following are the benefits provided by the framework:

- ***Dynamic installation of applications:*** The framework enables users to deploy/update the functions composing their applications on the devices efficiently and without being involved in tasks related to the configuration or the provisioning of the infrastructure.
- ***Fleet Deployment:*** Fleet deployment is directly connected with the previous item. This mechanism is supported by the S4T components that aggregate a group of devices as a fleet and manage it as a whole and by the FaaS Qinling subsystem with a mechanism to inject a function on multiple nodes (see Section 5.2).
- ***Optimized performances:*** by pushing IoT-related tasks (e.g., actuation decisions) to the network Edge, issues coming from services' latency are reduced due to the proximity of execution to the end device, and the security of the overall system is strengthened by data transmission via virtualized private networks (realized by WebSocket). Moreover, a function can react to particular events so that the devices are exempted from sending continuous data to the Cloud. This way, the bandwidth consumption is reduced.
- ***Automatic scaling:*** By design, a FaaS system can scale up/down depending on the number of functions execution requests on a node. This is obtained via the automatic orchestration of containers.
- ***OpenStack APIs:*** The proposed framework, even if it is easily exploitable by a user through Node-RED, is compliant with the OpenStack API structure, and in more detail, it is primarily exploitable via Qinling APIs, which can activate all mechanisms involving the other subsystems.

5. System design

In order to define the framework enabling the exploitation of FaaS facilities by the I/Ocloud, this section analyzes the extension of OpenStack subsystems (namely Qinling, Zun, IoTronic) aiming to manage functions at the network Edge. Concerning the FaaS coordination, Qinling and Zun were modified and integrated with IoTronic to provide suitable features related to IoT management. In the following, we report the modifications and the integration made to design the Edge-based FaaS system.

5.1. Edge-based FaaS containerization with Zun

In standard OpenStack Cloud deployments, the FaaS subsystem, Qinling, uses, by default, Kubernetes, or it can be configured to use Docker Swarm as COE. However, in both cases, the administrator has to manage the clusters manually, making such solutions hard to use in environments with high dynamicity. This is the case for IoT deployments, where IoT devices can be added to or removed from the deployments. To overcome this limitation and reduce the problems coming from integrating intrinsically different systems, the framework infrastructure exploits only OpenStack projects, so full compatibility is granted among all the subsystems. Moreover, exploiting this compatibility simplifies the resulting configuration duties for the administrator. For this purpose, we have extended the Zun subsystem capabilities to provide, for Qinling, a built-in orchestrator compliant with OpenStack. Consequently, in a straightforward manner, the built-in Zun orchestrator will use the Zun Compute nodes as its cluster. When a new IoT device (i.e., a Compute node) is added to the deployment, the system automatically includes it within the cluster.

Regarding the reachability of the runtime containers, in Cloud-only deployments, Zun relies on the networking technologies provided by OpenStack, such as Neutron/Kuryr, to make the containers/runtimes reachable based on a private IP addressing. In the Edge-based FaaS deployments, since the IoT devices are deployed outside the Cloud, may not have routable IP addresses, and can even be deployed behind NATs, reaching the runtimes they can host is not taken for granted as the case when considering standard Cloud deployments. For this purpose, we developed for Zun a new networking driver that exploits IoTronic to expose the capsules (i.e., the runtime containers) to the users through the use of the service forwarding capability provided by IoTronic (see Section 3.5). A runtime deployed on an IoT device will be exposed using a Cloud public IP address and a specific port. For the containers' *Permanent Volumes*, we conceived our FaaS system in a way to deploy the containers using Docker Local Volumes to make their storage non-volatile. We developed for this purpose a new Zun volume driver. A major difference between a standard FaaS system and an IoT-related one is the selection policies used to handle the device where a function has to be deployed. While analyzing the Zun-scheduler's selection rules, we noticed no option to request Zun to create a container/*capsule* on a specific node. For containers/*capsules* instantiation, the Zun scheduling procedure is done in two steps:

- ***Filtering:*** This step finds the set of nodes where deploying a container/*capsule* is feasible based on a set of filtering rules
- ***Scoring:*** This step assigns a score to any nodes that have passed the first step and choose the most suitable.

The filtering rules used by the standard Zun container manager are (i.e., scheduler):

- ***CPUFilter:*** The nodes are filtered according to the portion of CPU (free/allocated). The node with the highest free portion is the most preferred for the deployment.
- ***RAMFilter***: Similar to the *CPUFilter* rule, the *RAMFilter* is based on memory utilization.
- ***LabelFilter:*** In this case, nodes with a particular label node are ranked higher than others.
- ***ComputeFilter:*** A simple filter returns any host whose compute service is enabled and operational.
- ***AvailabilityZoneFilter:*** This filter uses the OpenStack availability zone option to select the compute node.
- ***DiskFilter:*** In this case, the hosts are sorted based on the free disk space with the largest weight winning.

From these filtering rules, we can notice that a scheduling policy based on the *hostname* is missing. Consequently, the user cannot request the creation of a runtime on a specific node. This rule is crucial for our use case as users will have more control over the deployment and, thus, be able to select an Edge device where a function/runtime should be deployed. For this purpose, we added a new filter for Zun based on the hostname we called *HostnameFilter*.

5.2. Edge-based FaaS with Qinling

The Qinling subsystem was also extended to meet the requirements of the FaaS for the I/Ocloud framework. In particular, as the framework includes Zun as an orchestrator, Qinling was extended to use the built-in Zun orchestrator instead of Kubernetes or Docker Swarm. Furthermore, to enable Zun to schedule a particular IoT device based on its hostname or a group of devices using a specific label, Qinling should specify in its requests to Zun the scheduling filters to be used in the selection: the *HostnameFilter* and *LabelFilter*. Here, we modified the Qinling structure by introducing two new features:

- **nodeName**: The nodeName is an attribute used to create a runtime in a node with a specific hostname. The Zun scheduler will use the *HostnameFilter* filtering rule in this case.
- **nodeSelector**: The **nodeSelector** is an attribute to create runtimes in a pool of nodes with one or many specific labels. The Zun scheduler will use the *LabelFilter* filtering rule in this case.

Until now, the new features introduced in Qinling are related to creating runtimes. However, for the execution of a function, more is needed. Indeed, let us consider standard Qinling implementations in Cloud environments to identify the runtime where a function should be executed. Qinling relies on the OpenStack networking service's IP address (i.e., Neutron/Kuryr). In our case, the Edge-based capsules cannot use standard OpenStack networking services, so it is impossible to deal with the particularity of IoT deployments (Neutron does not manage distributed instances deployed outside the Cloud). Furthermore, it cannot exploit a private IP address to select a runtime. To manage the association between the incoming execution requests and the runtimes, they are associated with the couples {Cloud IP address, port}. Hence, each runtime deployed at the network Edge is reachable through the Cloud (public) IP address and a specific port. The reverse proxy routes the requests received towards the right runtime on the IoT devices' side. In particular, the request going from Qinling to the Edge-based devices is modified by adding a new field, *runtime_id*, pointing to the IoT device's correct runtime. Once the reverse proxy receives a request, it checks the *runtime_id* field contained in the request and associates it with one of the capsules it manages (each pause container has a label called *runtime_id*). Of course, a modification to the Qinling database becomes necessary to store the capsules' *runtime_id*s as well.

5.3. Workflows

The following describes two important workflows that our FaaS for the I/Ocloud framework exploits. Specifically, we report the basic actions when a user requests the creation of a runtime on a specific IoT device and the workflow when a function is being requested to be executed on the runtime already created. To make the descriptions easy to follow, we report a simplified version of the workflows without taking

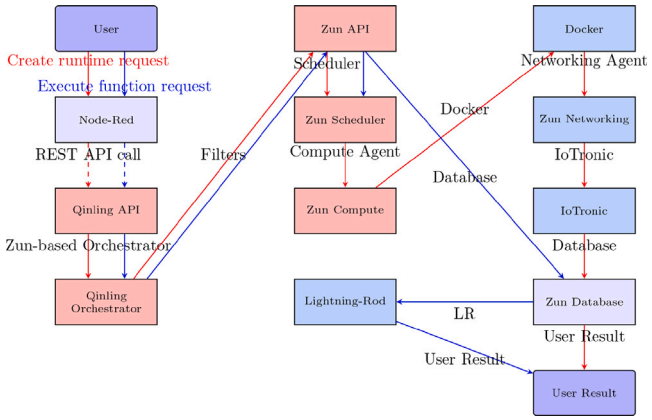

Fig. 5. Runtime creation and Function execution workflows.

into account some (default) OpenStack procedures: the ones related to Keystone duties to authenticate/authorize the incoming requests and the other OpenStack procedures based on tools such as Etcd to store information or RabbitMQ to dispatch events. In particular, we spotlight the interactions among the three main subsystems (i.e., Qinling, Zun, and IoTronic). This analysis also excludes the IoT registration phases. Fig. 5 shows a schematic, fully comprehensive direct acyclic graph representing the two workflows analyzed in detail in the following subsections.

5.3.1. Runtime instantiation workflow

A function execution in a FaaS system needs an opportune runtime instantiated in the device designed to execute the designed function; here is provided a detailed description of the interaction involving the main elements of the framework, and furthermore, a graphical view of this workflow is depicted both in Fig. 5 and in the activity diagram in Fig. 6. According to the presented framework purposes, the described workflow aims to instantiate a runtime on a device deployed at the network Edge:

1. The user requests to deploy a runtime on a specific remote node through the dashboard or the CLI. Then, the dashboard/CLI performs a specific REST request to the Qinling-API server.

2. The Qinling-API server forwards the request to create the runtime to the Qinling-orchestrator driver designed to interact with Zun.

3. The Qinling-orchestrator sends the request to the Zun-API server with a particular body specifying the *nodeName* attribute (i.e., the hostname of the device where the runtime should be created. See Section 5.1).

4. The Zun-API server forwards the request to the Zun scheduler to identify the host (i.e., IoT device) where the runtime should be created.

5. The Zun scheduler applies the new filter *HostnameFilter* (see Section 5.2) to schedule the appropriate host. Afterward, it sends a request to create a *capsule* to the Zun-Compute agent running on that host.

6. After receiving the request, the Zun-Compute agent sends an HTTP request on localhost to the Docker engine to create the *capsule*/containers.

7. Docker creates the containers needed.

8. After creating the containers, the Zun-compute agent operating in the framework requests the Zun networking driver, which uses IoTronic functionalities, to expose the *capsule* (i.e., the pause container) to the users.

9. The Zun networking driver interacts with the IoTronic subsystem to expose the *capsule*. More in detail, IoTronic exposes the capsule through a particular port associated with a public IP address on the Cloud. Then, a WS tunnel is created between the IoT device and the Cloud. Hence, any request received on the Cloud port/IP address will be forwarded through the WS tunnel to a port already opened on the IoT device (configured to reach exactly the container of a specific runtime linked to a function execution. The reverse proxy exploits this port).

10. IoTronic sends back to the Zun networking driver the port and the Cloud IP address associated with the *capsule*; then, the metadata of the capsule already created in step 7 (i.e., capsule *runtime_id*, IP/port) will be stored in the Zun database.

11. The Zun-Compute on Cloud sends a notification about the operation status to the Zun-scheduler. The response contains the *runtime_id* of the capsule created that will be stored in the Qinling database (the *runtime_id* is essential to reach the runtime when a function should be executed).

12. The Zun-scheduler forwards the notification to the Zun-API server that contacts, in its turn, the Qinling subsystem to store on its database the *capsule runtime_id*.

The workflow reported previously concerns to the runtime instantiation on a single IOT device. In order to deploy runtime on several IoT devices having the same label, the same workflow will take place while specifying in step 3 the *nodeSelector* attribute instead of *nodeName*. Therefore, the Zun scheduler will use the *LabelFilter* to select the set of IoT devices where the runtime should be deployed.

5.3.2. Function execution workflow

The workflow related to a function execution on a specific runtime deployed on an Edge node using our FaaS for the I/Ocloud framework is depicted in Fig. 5 and in the activity diagram in Fig. 7.

As an essential requirement, we assume that a hosting node (i.e., an IoT device) is already registered to the Cloud. The user has already written his/her function and associated it with a particular runtime (so the *function_id* is stored on the Qinling database). As the last prerequisite, the runtime is already deployed on the Edge device. The following steps describe the workflow of executing the function on the runtime:

1. The user sends a request to execute a function on a specific IoT node, either through the dashboard or the CLI, and according to the above assumptions, the runtime is already deployed on the remote node. Then, the dashboard/CLI performs a specific Qinling API call via REST. The request body contains the *function_id* and the *nodeName* to identify the IoT device where the function has to be executed (or the *nodeSelector* in case of function to be executed on group IoT devices).

2. After retrieving the involved runtime_id from its database, the Qinling-API server forwards the request to the Qinling orchestrator driver (meant to interact with Zun).

3. The orchestrator contacts Zun using the API server to get the metadata of the runtime involved (i.e., IP/port used by IoTronic to expose the runtime).

4. The Zun-API server forwards the request to the Zun database to retrieve the information.

5. The Zun database sends back the metadata to the Zun-API server.

6. The Zun-API server forwards back the response to the Qinling orchestrator.

7. Based on the metadata received, the orchestrator routes the execution request through the WS tunnel previously created when the runtime was deployed on the Edge node. The request forwarded contains a field with the *runtime_id*.

8. Once the request reaches the IoT device through the WS tunnel, LR uses the reverse proxy to identify the runtime concerned. Specifically, the reverse proxy uses the *runtime_id* field in the received request to make the association.

9. The result is returned to LR once the function has been executed on the runtime.

10. LR then sends the result back to the Qinling-Orchestrator via the WS tunnel.

11. The orchestrator sends the result to the Qinling-API server.

12. Finally, the function execution result is returned to the user or shown on the dashboard according to the workflow defined.

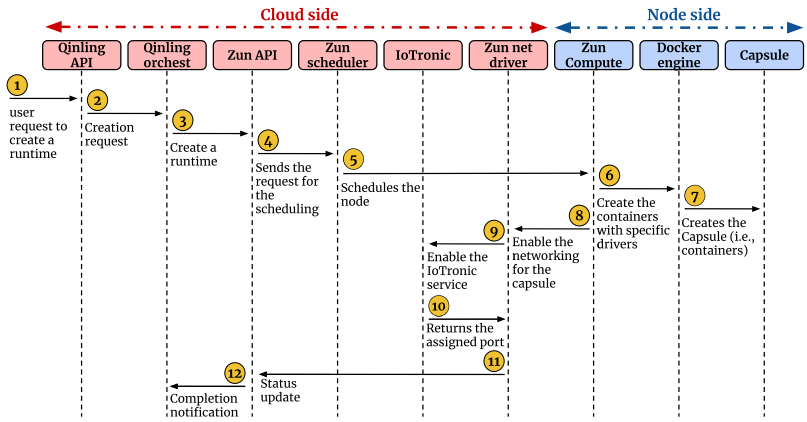

Fig. 6. A runtime instantiation workflow.

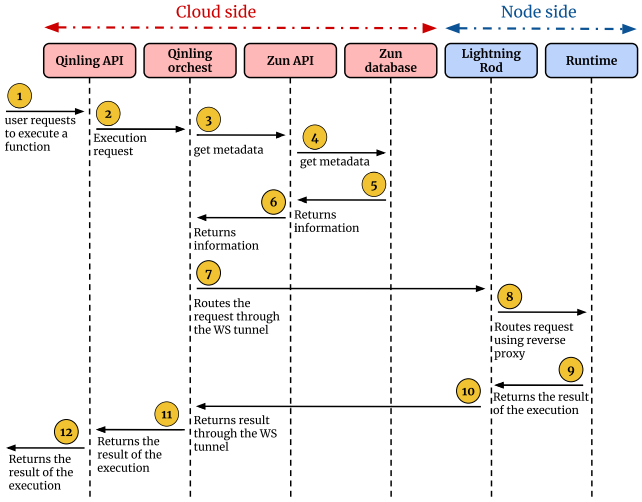

Fig. 7. A function execution workflow.

The aforementioned workflow is also used when the function has already been executed at least once.

When a function is invoked, the runtime container can load the necessary packages from the shared volume that the sidecar container has prepared during the 'Runtime Creation Workflow'. However, additional operations must occur when a function is invoked for the first time. Specifically, once the request reaches the IoT device involved (step 8), the relevant *capsule* must retrieve the packages of the invoked function from the Cloud and store them on the *Persistent Volume*. This goal is obtained by the sidecar container communicating with the Cloud to download and store the necessary packages locally. Once this is done, the runtime container can load and execute the function.

6. Use case

Our system enables users to create and deploy applications on remote IoT devices or gateways. These devices, in turn, are designed to respond swiftly to specific circumstances, mitigating the dependence on remote Cloud services.

Fig. 8 presents a detailed use case of Monitoring and Control Systems deployment in a Multi-line Production Facility. Here, a production controller[6] creates an application to monitor and control processes in a production facility with multiple production lines, where the operational temperature significantly influences the final product quality. These processes may encompass several procedures, such as forging, turning, milling, and drilling. The application, once designed, is actualized using a user-friendly drag-and-drop interface on the Node-RED dashboard. The deployment of the flow involves a series of procedures intended to initiate runtimes and execute functions on the selected devices, as enabled through the FaaS approach within the I/Ocloud framework.

Each production line in this use case is supported by an external refrigeration system that responds when the temperature escalates too quickly to be controlled by the internal cooling system. The additional temperature control layer demonstrates the system's practical application in real-world scenarios.

The analysis of this use case can be broken down into three phases: (i) application definition, (ii) application deployment and execution, and (iii) event detection.

Fig. 8(a) depicts the initial state where an administrator defines or modifies the monitoring and control operations application. To accomplish this task, the administrator interacts with the Node-RED dashboard, where a set of predefined nodes – each representing an atomic function – can be configured and interconnected to formulate the application workflow.

The ease inherent in deploying functions via the FaaS framework makes it an ideal programming model for this scenario. In the subsequent phase, illustrated in Fig. 8(b), the administrator deploys the Node-RED flow, generating a series of requests for runtime creation and function injection on the devices outlined by the appropriate configuration parameters (i.e., nodeSelector or nodeName) defined in the initial phase. Upon successfully completing the deployment, the administrator can activate the injected functions.

During the final phase, while executing the functions, a function may interact with another function in response to an event. This interaction uses the execution parameters obtained by Qinling from a query conducted by the Node-RED dashboard during the execution phase.

Figs. 8(a), 8(b), and 8(c) illustrate a scenario involving four production lines, each outfitted with an injected monitoring function. In the event of overheating, this function triggers a cooling action executed by another function housed within the external refrigeration system.

7. Approach and system evaluation

The evaluation of the proposed approach and framework is carried out over many perspectives, starting from an evaluation of the impact on the end device-side scalability and stepping through an analysis of the overhead introduced by the FaaS paradigm concerning the traditional approach of handcrafted setup for the component of a distributed application. To complete our analysis, we compare an application tailored to the use case shown in Fig. 8.

[6] Commonly, the production controller is also called *production manager* or *manufacturing manager*.

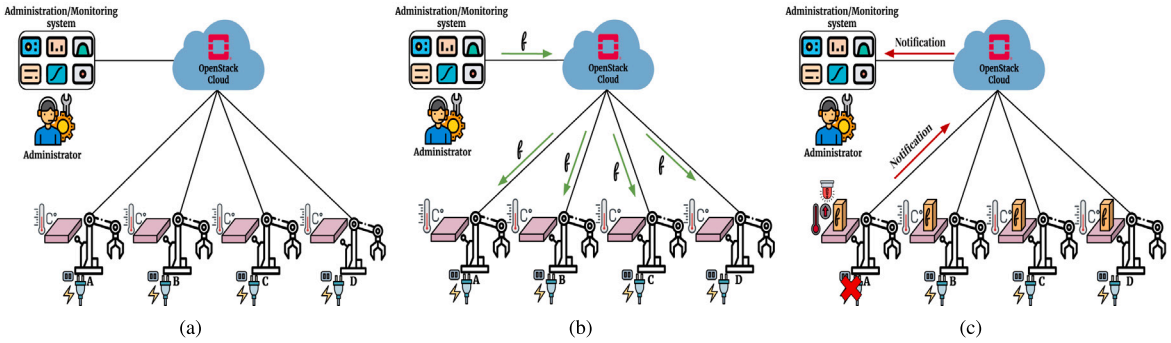

Fig. 8. Overview of a use case where the Edge-based FaaS system can be deployed for better management tasks.

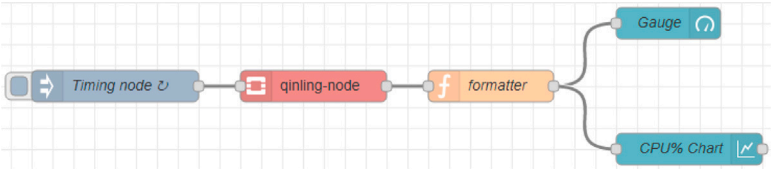

Fig. 9. Node-RED flow involving a Qinling-node.

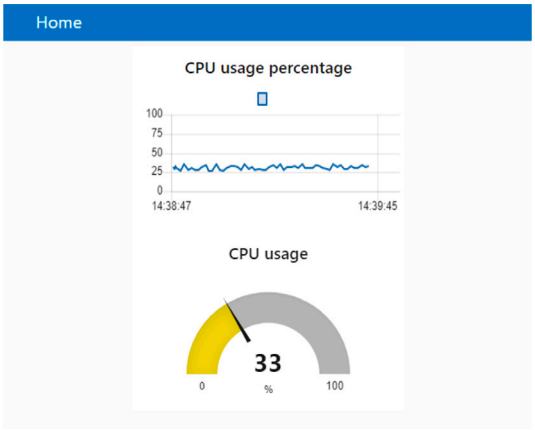

Fig. 10. Flow output on the Node-RED dashboard.

7.1. Scalability of FaaS on IoT nodes

The first step in the evaluation is understanding how the FaaS approach impacts devices in the I/Ocloud. The framework can exploit IoT boards as (at least) compute nodes hosting application componentry according to the flow defined in the NodeRED dashboard. A user of the framework can obviously expose multiple functions of a single node (i.e., multiple instances of the same function related to one or multiple flows or multiple instances of different functions). For this reason, we will analyze the behavior of two characteristics discussed above in Section 4.5: (i) Automatic scaling and (ii) Optimization of performance and reliability.

The analysis has been carried out on a simple monitoring function that reads a value and delivers it to the FaaS system. Fig. 9 shows the flow used to implement the analysis logic; it can deploy multiple instances of a function defined in the "qinling-node", ready to be executed on the same IoT node. After the deployment and execution of these functions, the NodeRed system is responsible for visualizing the outcome. Fig. 10 captures this aspect, illustrating the NodeRed dashboard's flow output in terms of CPU usage percentage. This Figure provides a visual representation of the results as they are returned from the executed functions on the IoT device, highlighting the prompt feedback provided by NodeRed. The test aims to measure the time needed to complete requests under different FaaS platform settings for concurrency; in particular, each test set runs with a different load-balancing policy for Qinling. According to each configuration, the runtimes deployed on the board manage 2 to 8 concurrent requests before an additional runtime is spawned. Tests are defined in groups of 30 attempts per working condition, then reported in terms of their averages. The testbed is composed of a few Virtual Machines plus an emulated IoT device with 1 GB of RAM and one vCPU. Fig. 11 shows the most significant three out of seven behaviors (i.e., 2-to −8 concurrency levels) emerging from the tests; in particular, Fig. 11 represents the best behavior, alongside the two worst ones.

The case of 2 concurrent requests admitted had the most uneven behavior because, as requests increase, runtimes spawned on the device also grow at a rapid pace (i.e., every other concurrent request), saturating the resources available on the device. The peaks, indeed, are aligned to the spawning of additional runtimes. The case of 8 concurrent requests admitted on the same device, as depicted in the figure with a yellow line, has less nonlinear behavior. According to this graph, the best configuration (i.e., less elapsed time, better concurrent request management) is the one drawn in red, representing a concurrency of 7 requests. During the experiments, a certain number of failures[7] have been observed. Fig. 12 shows the relationship between the percentage of failures vs the total number of incoming (concurrent) requests. In the configuration with 7 concurrent requests, depicted with a red line, we observe the most favorable behavior, in line with our remarks on the previous graph. Specifically, the rate of failures begins to rise once a (comparatively) higher number of concurrent requests is reached, moreover, at a slower pace compared to other configurations.

Another aspect under analysis is the behavior of the system when the number of functions deployed on the IoT node grows. Indeed, the execution of an additional function on the same device requires a new capsule at the edge (e.g., on a constrained IoT device), corresponding to (at least) three containers created as described in 3.4. Fig. 13 shows the elapsed service times on average where either one or two functions get deployed by the FaaS platform on a single node. Results highlight the higher performance achieved in the latter case, i.e., the number of

[7] A failure happens when a timeout is reached; the testbed is configured with a connection timeout equal to 3.05 s and a response timeout equal to 60 s.

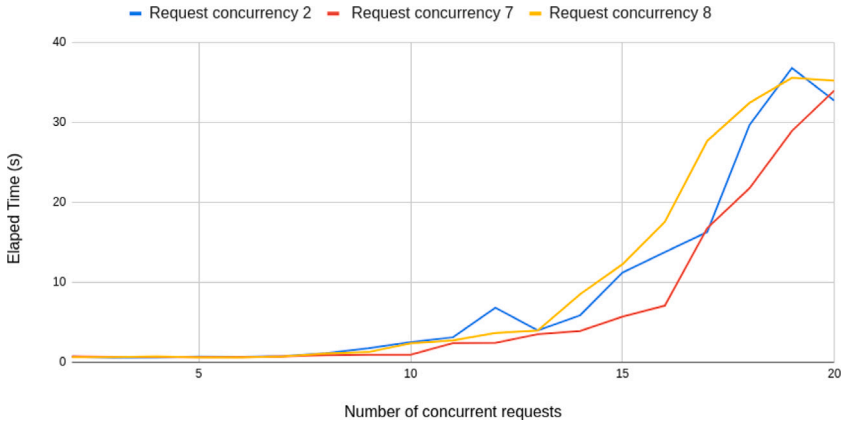

Fig. 11. Elapsed time to execute one simple function with three different concurrency configurations.

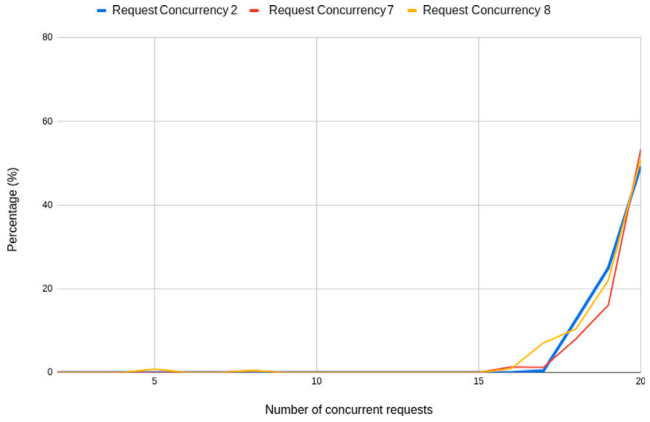

Fig. 12. Failures occurred in the execution of simple function with three different concurrency configurations.

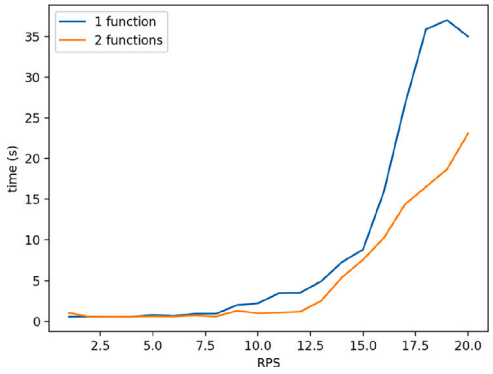

Fig. 13. Average elapsed times vs RPS.

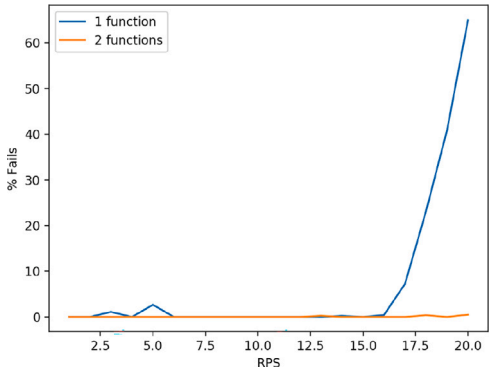

Fig. 14. Percentage of failed executions vs RPS.

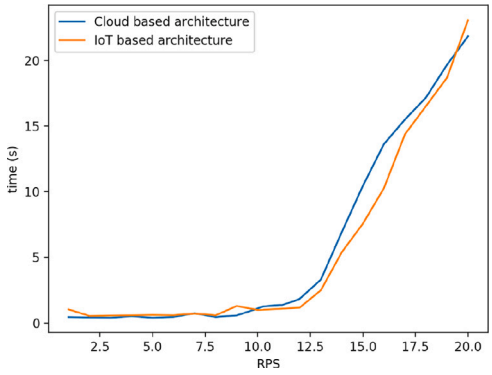

Fig. 15. Impact of Datacenter-only vs. IoT-oriented drivers on networking performance.

requests per second (RPS) that a node can handle before response times degrade significantly by hosting multiple functions; thus, it is beneficial to devise very scope-limited functions on a single node instead of designing (e.g., monolithic) functions, i.e., too rich in duties to fulfill. This approach may lead to a comparatively higher number of requests each node can manage with respect to the number of functions deployed. The results show again how an approach using more scope-limited functions is better in terms of performance.

Accordingly, corresponding failures are depicted in Fig. 14.

Another round of testing was meant to evaluate the impact of leveraging IoTronic facilities for network management, as shown in Fig. 15, comparing the behavior of our platform that uses IoTronic as a networking driver (we refer to this scenario, in the figure, as IoT-based architecture) and a (vanilla) Datacenter-only FaaS deployment where a Flannel driver[8] is used to manage the containers' networking facilities (referred to as Cloud-based architecture in the figure). The performance of the two scenarios is comparable. Consequently, using IoTronic as a networking driver does not seem to impact system performance measurably.

[8] The Flannel driver is the default driver used by Kubernetes, ZUN, and other Container Orchestration Environment to manage the networking among containers as an overlay network. It offers an IP for each container.

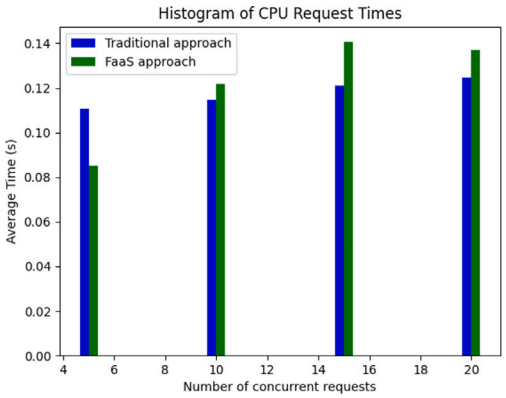

Fig. 16. Single-function invocation performance: script vs function (as in FaaS) comparison.

7.2. FaaS overhead

The overhead introduced by the FaaS platform is analyzed to measure its impact. This way, the key performance indicators in the previous scenario are measured and compared with the case in which a simple HTTP server script is running on a device and performs what is set in Fig. 9 with the FaaS: on a device, an HTTP server is set up with a function able to read the CPU serving parallel requests by exploiting a multi-threading approach. The end-to-end response times are compared in Fig. 16 with data originating from the previous analysis to evaluate system overhead. The comparison highlights a slight performance degradation when concurrent requests keep growing due to multiple runtimes being instantiated into a device to manage independent incoming requests. Nevertheless, the difference is minimal, and there is even a crossover point all the way to the left of the graph, i.e., at lower concurrency levels, beyond which (e.g., around 5 and below) FaaS actually provides an edge in terms of performance.

7.3. Comparison with a conventional development workflow

The last step that completes the analysis of the proposed solution is performed to analyze its impact on the application developer. To cope with this goal, a further comparison is made in terms of coding duties up to a developer: firstly, we implement a dummy application representing the Use Case shown in Fig. 8 where a client application runs on the production line and in case of an event invokes a server application that controls the external refrigeration system, to actuate according to the cooling request, and secondly, we go through an analysis of steps requested to define the application with and without our solution.

The first comparison draws on the code listings below, per the following description: the dummy application devised without resorting to deviceless FaaS is shown in Algorithms 1 and 2; conversely, the dummy application designed on top of our FaaS approach is shown in Algorithms 3, 4,5, 6, and 7. Nevertheless, by making a straightforward comparison in terms of LOC, i.e., Lines of Code, it is evident that the FaaS-based solution reduces the quantity of code necessary for the developer (48 LOC versus 114 LOC in the case of the standalone implementation). It is worth noting that although the code listings, as mentioned earlier, exemplify the FaaS approach, these do not factor in flow definition duties within the Node-RED dashboard, as needed to set up the application. Indeed, in order to replicate the application faithfully, two additional custom nodes (e.g., built beforehand) are needed, one for the *Server* side and another one for the *Client* side, obtained through a simple drag&drop action of the selected node and its suitable configuration.

A more thorough overview of the advantages provided by the proposed approach to developers may be outlined by considering the steps needed to implement the aforementioned application. Nevertheless, despite the issues in evaluating how demanding coding may be overall, we are confident that this outline provides a rough estimate of the effort involved in the process. Figs. 17 and 18 show a possible workflow of application definition involving four identical production lines connected to an external on-demand cooling system. By analyzing the two activity diagrams, it is clear that the end-device interaction in the FaaS approach relies totally on Node-RED, conversely from the traditional approach, which requests an interaction with every system involved. Another meaningful difference is represented by the configuration steps named "*Production Line Identification*" corresponding to creating the four files used in Algorithm 2 to enable the external cooling system to move towards the right production line. Moreover, the configuration steps done with the FaaS approach are safer because they are done once in the Node-RED dashboard instead of repeating them in each *Production Line* part of the system, as shown in Fig. 19. Finally, the FaaS approach, due to the exploitation of a specific runtime for function execution, decouples the runtime environment from function dependencies

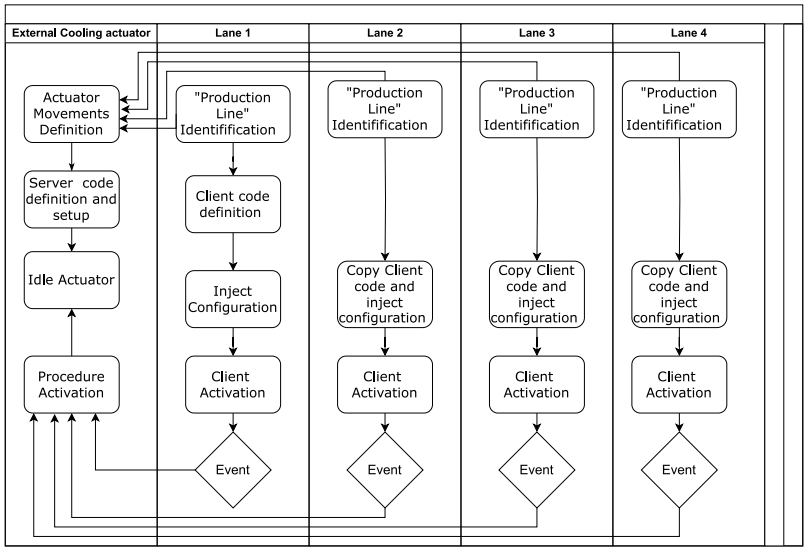

Fig. 17. Dummy application setup steps (traditional approach).

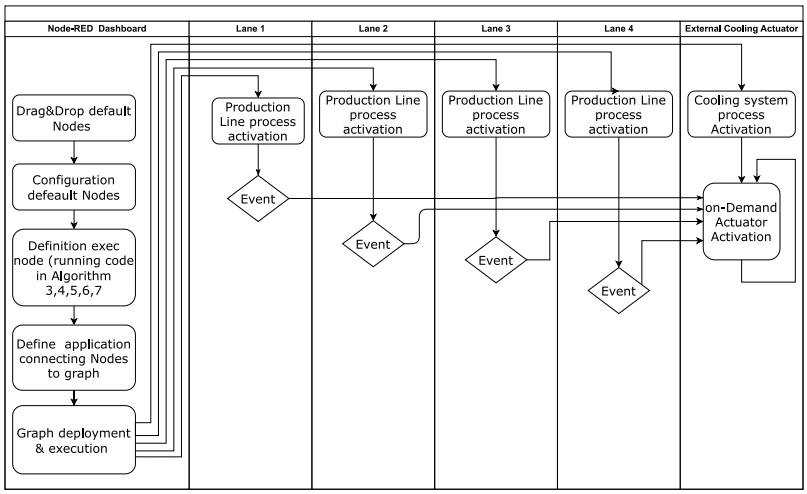

Fig. 18. Dummy application setup steps (FaaS approach).

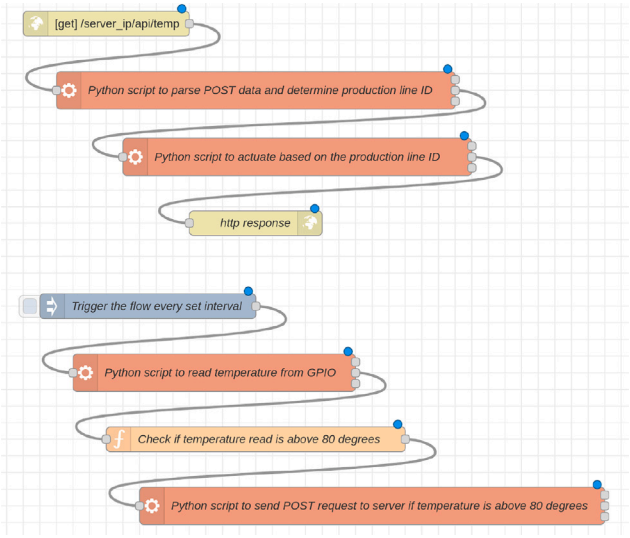

Fig. 19. Dummy application in Node-RED.

(e.g., libraries, settings) thanks to the isolation offered by the capsule environments.

8. Conclusion and future works

In this research, we presented a Function-as-a-Service platform designed to manage and program Internet of Things resources efficiently. This platform allows the definition and execution of function-based pipelines at the network's Edge, encompassing Fog or Edge devices. The solution features a dashboard providing a graphical interface for an integrated development environment to aid IoT developers in designing data pipelines. Our preliminary tests prove that the proposed solution accommodates scenarios where extensive use of IoT-hosted resources is expected. Metrics of usage patterns include scope/purpose, such as instantiating multiple functions as part of one or more pipelines and sharing among multiple users, such as managing concurrent invocations. We aim to incorporate Fog/Edge devices management using the FaaS facilities outlined in this work. This approach will better define Fog computing's role as a coordinator of (e.g., clusters of) IoT devices and/or resources involved in specific pipelines, providing system administrators with the chance to define a function as a planned reaction to events. Moving forward with this research, we expect to refine and expand the capabilities of our FaaS platform, gauging its applicability and optimizing its effectiveness in various IoT scenarios.

CRediT authorship contribution statement

Giovanni Merlino: Conceptualization, Methodology, Project administration, Supervision, Writing – review & editing. **Giuseppe Tricomi:** Conceptualization, Investigation, Methodology, Writing – original draft, Writing – review & editing. **Luca D'Agati:** Visualization, Writing – review & editing. **Zakaria Benomar:** Writing – original draft. **Francesco Longo:** Project administration, Supervision. **Antonio Puliafito:** Project administration, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work is partially supported by "SoBigData.it receives funding from European Union – NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) – Project: "SoBigData.it – Strengthening the Italian RI for Social Mining and Big Data Analytics" – Prot. IR0000013 – Avviso n. 3264 del 28/12/2021."

Appendix. Algorithms

See Algorithms 1 to 7 and Table 1.

Algorithm 1: Fragment of a standalone code to control the temperature on the product line

```
import time
import requests
import RPi.GPIO as GPIO
SERVER_URL = 'http://X.X.X.X:8000'
my_variable=0;
prodlineid = 0
;   /* prodlineid variable represents the ID of the production line where this code runs */
interval = 2;
pin = 17
;                  /* This code suppose to use DHT11 or the DS18B20 */
def send_post_request(value,prodlineid):
;                    /* Function to request refrigerator actuation */
    payload = { 'campo': str(value), 'line_number': str(prodlineid) }
    try:
        response = requests.post(SERVER_URL, data=payload)
        print("Request of cooling sent to device. Response received:", response.text)
        response.close()
    except request.exceptions.RequestException as e:
        print("Error sending request:",e)
;                                          /* GPIO initial configuration */
def read_temperature():
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(pin, GPIO.OUT)
;                  /* The communication with the sensor begins here */
    GPIO.output(pin, GPIO.HIGH)
    time.sleep(0.1)
    GPIO.output(pin, GPIO.LOW)
    time.sleep(0.018)
;                                 /* Configure GPIO pin to read sensor  */
    GPIO.setup(pin, GPIO.IN, GPIO.PUD_UP)
;                                                    /* Read sensor value */
    while GPIO.input(pin) == GPIO.LOW:
        pass
    start_time = time.time()
    while GPIO.input(pin) == GPIO.HIGH:
        pass
    end_time = time.time()
;       /* Computing temperature with respect to pulse duration  */
    pulse_duration = end_time - start_time
    temperature = pulse_duration * 1000
    GPIO.cleanup()
    return temperature
while True:
    temp = read_temperature()
;     /* Critical condition requesting an extra cooling activity */
    if temp > 80:
        send_post_request(my_variable,prodlineid)
    time.sleep(interval)
```

Algorithm 2: Fragment of a code to control the external cooling device

```
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
from urllib.parse import urlparse, parse_qs
import threading
semaforo = threading.Semaphore(1)
class MyHandler(BaseHTTPRequestHandler):
    def on_demand_actuation(self,line_number):
;     /* The code is referring to the use case reported in Fig. 8  */
        if line_number == 1:
            nome_file = 'alpha.cfg'
        elif line_number == 2:
            nome_file = 'beta.cfg'
        elif line_number == 3:
            nome_file = 'gamma.cfg'
        elif line_number == 4:
            nome_file = 'delta.cfg'
        else:
            raise ValueError("Value not admitted. Production line ID not identified.")
        self.move_actuator(nome_file)
    def move_actuator(self, nome_file):
        try:
            if semaforo.acquire(blocking=False):
                with semaforo:
;       /* The Semaphore usage is needed to avoid the concurrential access on actuator */
                    with open(nome_file, 'r') as file:
                        for row in file:
;     /* Files represent a list of rows containing the instructions necessary to reach a production line from the base position */
                            command = row.strip()
                            print(command)
                            self.executeCommand(command)
            else :
                semaforo.acquire()
                with semaforo:
                    with open(nome_file, 'r') as file:
                        for row in file:
                            command = row.strip()
                            print(command)
                            self.executeCommand(command)
        except FileNotFoundError:
            raise FileNotFoundError(f"The file {nome_file} does not exists.")
    def executeCommand(self, command):
;                                    /* function logic not implemented  */
        return
    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length).decode(' utf-8 ')
        parsed_data = parse_qs(post_data)
        if 'prodline' in parsed_data:
            prodline = parsed_data['prodline'][0]
            print("Production Line requesting actuation is:", prodline)
        try:
            self.on_demand_actuation(prodline)
        except ValueError as e:
            print(str(e))
            self.send_response(500)
            return
        except FileNotFoundError as ef:
            print(str(e))
            self.send_response(500)
            return
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()
        response_message = 'Cooling action done on prod. line with id: '+prodline
        self.wfile.write(response_message.encode('utf-8'))
class ThreadedHTTPServer(ThreadingHTTPServer):
    def process_request(self, request, client_address):
        thread = threading.Thread(target=self._new_request_thread, args=(self, request, client_address))
        thread.start()
    def _new_request_thread(self, server, request, client_address):
        server.ProcessRequestThread(request, client_address)
    def ProcessRequestThread(self, request, client_address):
        self.finish_request(request, client_address)
        self.shutdown_request(request)
def run(server_class=ThreadedHTTPServer, handler_class=MyHandler, port=8000):
    server_address = ('X.X.X.X', port)
    httpd = server_class(server_address, handler_class)
    print(f"Server is reachable at port {port}")
    httpd.serve_forever()
run()
```

Table 1
Table of acronyms.

Acronym	Meaning
API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend-as-a-Service
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
COE	Container Orchestration Environment
CPS	Cyber-Physical Systems
Cgroups	Control Groups
FaaS	Function-as-a-Service
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IoT	Internet of Things
IT	Information Technology
K8s	Kubernetes
LAN	Local Area Network
LOC	Lines of Code
LR	Lightning-Rod
NAT	Network Address Translation
OS	Operative System
PaaS	Platform-as-a-Service
QoE	Quality of Experience
QoS	Quality of Service
REST	Representational State Transfer
RPC	Remote Procedure Calls
RPS	Requests per Second
S4T	Stack4Things
Seccomp	Secure Computing Mode
UI	User Interface
VMs	Virtual Machines
VN	Virtual Node
vCPU	Virtual Central Processing Unit
WAMP	Web Application Messaging Protocol
WS	WebSocket

Algorithm 3: Node-RED ‘exec‘ node used to execute the Python code related to the second node of the graph *Client*

```
import RPi.GPIO as GPIO
import time
def read_temperature(pin):
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.HIGH)
    time.sleep(0.1)
    GPIO.output(pin, GPIO.LOW)
    time.sleep(0.018)
     GPIO.setup(pin, GPIO.IN, GPIO.PUD_UP)
     while GPIO.input(pin) == GPIO.LOW:
         pass
     start_time = time.time()
     while GPIO.input(pin) == GPIO.HIGH:
         pass
     end_time = time.time()
     pulse_duration = end_time - start_time
     temperature = pulse_duration * 1000
     GPIO.cleanup()
     return temperature
```

Algorithm 4: Node-RED ‘exec‘ node used to execute the Python code related to the third node of the graph *Client*

```
def check_temperature(temperature):
    if temperature > 80:
        return True
    return False
```

Algorithm 5: Node-RED ‘exec‘ node used to execute the Python code related to the forth node of the graph *Client*

```
import requests
SERVER_URL = 'http://X.X.X.X:8000'
def send_post_request(value, prodlineid):
    payload = 'campo': str(value), 'line_number': str(prodlineid)
    response = requests.post(SERVER_URL, data=payload)
    return response.text
```

Algorithm 6: Node-RED ‘exec‘ node used to execute the Python code related to the second node of the graph *Server*

```
from urllib.parse import parse_qs
def parse_post_data(post_data):
    parsed_data = parse_qs(post_data)
    if 'prodline' in parsed_data:
        prodline = parsed_data['prodline'][0]
        return prodline
```

Algorithm 7: Node-RED ‘exec‘ node used to execute the Python code related to the third node of the graph *Server*

```
def on_demand_actuation(line_number):
    if line_number == 1:
        pnome_file = 'alpha.cfg'
    elif line_number == 2:
        pnome_file = 'beta.cfg'
    elif line_number == 3:
        pnome_file = 'gamma.cfg'
    elif line_number == 4:
        pnome_file = 'delta.cfg'
     else:
         raise ValueError("Value not admitted. Production line ID not identified.")
     return nome_file
```

References

[1] G. Adzic, R. Chatley, Serverless computing: economic and architectural impact, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, pp. 884–889.

[2] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, A. Ghalsasi, Cloud computing—The business perspective, Decis. Support Syst. 51 (1) (2011) 176–189.

[3] A.M. Joy, Performance comparison between linux containers and virtual machines, in: 2015 International Conference on Advances in Computer Engineering and Applications, IEEE, 2015, pp. 342–346.

[4] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, in: Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

[5] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, Serverless programming (function as a service), in: 2017 IEEE 37th International Conference on Distributed Computing Systems, ICDCS, 2017, pp. 2658–2659, http://dx.doi.org/10.1109/ICDCS.2017.305.

[6] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, O. Rana, Fog computing for the internet of things: A survey, ACM Trans. Internet Technol. (TOIT) 19 (2) (2019) 18.

[7] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, R. Prodan, A serverless real-time data analytics platform for edge computing, IEEE Internet Comput. 21 (4) (2017) 64–71.

[8] L. Baresi, D.F. Mendonça, Towards a serverless platform for edge computing, in: 2019 IEEE International Conference on Fog Computing, ICFC, IEEE, 2019, pp. 1–10.

[9] A. Botta, W. De Donato, V. Persico, A. Pescapé, Integration of cloud computing and Internet of Things: a survey, Future Gener. Comput. Syst. 56 (2016) 684–700.

[10] M. Díaz, C. Martín, B. Rubio, State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing, J. Netw. Comput. Appl. 67 (2016) 99–117.

[11] T. Lynn, P. Rosati, A. Lejeune, V. Emeakaroha, A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms, in: 2017 IEEE International Conference on Cloud Computing Technology and Science, CloudCom, 2017, pp. 162–169, http://dx.doi.org/10.1109/CloudCom.2017.15.

[12] A. Glikson, S. Nastic, S. Dustdar, Deviceless edge computing: Extending serverless computing to the edge of the network, in: In Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR, 2017.

[13] A. Alvarado, Serverless vs. FaaS: A beginner's guide, 2019, URL: https://www.liquidweb.com/kb/serverless-vs-faas-a-beginners-guide/.

[14] P. Johnston, Serverless: It's much much more than faas, 2018, URL: https://medium.com/@PaulDJohnston/serverless-its-much-much-more-than-faas-a342541b982e.
[15] L.E. Hecht, Add it up: FAAS ≠ Serverless, 2018, URL: https://thenewstack.io/add-it-up-serverless-faas/.
[16] CN.C.F. Serverless Working Group, Serverless Whitepaper V1.0, CloudNative Computing Foundation, 2018, URL: https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview.
[17] Apache, Apache OpenWhisk, 2019, URL: https://openwhisk.apache.org/.
[18] Microsoft, Azure IoT edge, 2019, URL: https://azure.microsoft.com/en-us/services/iot-edge/.
[19] Amazon, AWS greengrass, 2019, URL: https://aws.amazon.com/greengrass/.
[20] IBM, IBM watson IoT platform, 2019, URL: https://www.ibm.com/cloud/internet-of-things.
[21] G.A.S. Cassel, V.F. Rodrigues, R. da Rosa Righi, M.R. Bez, A.C. Nepomuceno, C. André da Costa, Serverless computing for Internet of Things: A systematic literature review, Future Gener. Comput. Syst. 128 (2022) 299–316, http://dx.doi.org/10.1016/j.future.2021.10.020, URL: https://www.sciencedirect.com/science/article/pii/S0167739X21004167.
[22] B. Cheng, J. Fuerst, G. Solmaz, T. Sanada, Fog function: Serverless fog computing for data intensive IoT services, in: 2019 IEEE International Conference on Services Computing, SCC, IEEE, 2019, pp. 28–35.
[23] P. Persson, O. Angelsmark, Kappa: serverless IoT deployment, in: Proceedings of the 2nd International Workshop on Serverless Computing, 2017, pp. 16–21.
[24] A. Garbugli, A. Sabbioni, A. Corradi, P. Bellavista, Tempos: Qos management middleware for edge cloud computing faas in the Internet of Things, IEEE Access 10 (2022) 49114–49127.
[25] P. Benedetti, M. Femminella, G. Reali, K. Steenhaut, Experimental analysis of the application of serverless computing to IoT platforms, Sensors 21 (3) (2021) 928.
[26] R. Wolski, C. Krintz, F. Bakir, G. George, W.T. Lin, Cspot: Portable, multi-scale functions-as-a-service for IoT, in: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, 2019, pp. 236–249.
[27] T. Pfandzelter, D. Bermbach, Tinyfaas: A lightweight faas platform for edge environments, in: 2020 IEEE International Conference on Fog Computing, ICFC, IEEE, 2020, pp. 17–24.
[28] R. Dua, A.R. Raja, D. Kakadia, Virtualization vs containerization to support paas, in: 2014 IEEE International Conference on Cloud Engineering, IEEE, 2014, pp. 610–614.
[29] Z. Kozhirbayev, R.O. Sinnott, A performance comparison of container-based technologies for the cloud, Future Gener. Comput. Syst. 68 (2017) 175–182.
[30] E.W. Biederman, L. Networx, Multiple instances of the global linux namespaces, in: Proceedings of the Linux Symposium, Vol. 1, Citeseer, 2006, pp. 101–112.
[31] R. Rosen, Resource management: Linux kernel namespaces and cgroups, 186, 2013, Haifux, May.
[32] M.A. Babar, B. Ramsey, Understanding Container Isolation Mechanisms for Building Security-Sensitive Private Cloud, Technical Report, CREST, University of Adelaide, Adelaide, Australia, 2017.
[33] F. Longo, D. Bruneo, S. Distefano, G. Merlino, A. Puliafito, Stack4Things: An OpenStack-based framework for IoT, in: 2015 3rd International Conference on Future Internet of Things and Cloud, 2015, pp. 204–211, http://dx.doi.org/10.1109/FiCloud.2015.97.
[34] S. Distefano, G. Merlino, A. Puliafito, Device-centric sensing: An alternative to data-centric approaches, IEEE Syst. J. 11 (1) (2017) 231–241, http://dx.doi.org/10.1109/JSYST.2015.2448533.
[35] D. Bruneo, S. Distefano, F. Longo, G. Merlino, A. Puliafito, I/Ocloud: Adding an IoT dimension to cloud infrastructures, Computer 51 (1) (2018) 57–65.
[36] G. Tricomi, Z. Benomar, F. Aragona, G. Merlino, F. Longo, A. Puliafito, A NodeRED-based dashboard to deploy pipelines on top of IoT infrastructure, in: 2020 IEEE International Conference on Smart Computing, SMARTCOMP, 2020, pp. 122–129, http://dx.doi.org/10.1109/SMARTCOMP50058.2020.00036.
[37] Z. Benomar, F. Longo, G. Merlino, A. Puliafito, Deviceless: A serverless approach for the Internet of Things, in: 2021 ITU Kaleidoscope: Connecting Physical and Virtual Worlds, ITU K, 2021, pp. 1–8, http://dx.doi.org/10.23919/ITUK53220.2021.9662096.
[38] Z. Benomar, D. Bruneo, S. Distefano, K. Elbaamrani, N. Idboufker, F. Longo, G. Merlino, A. Puliafito, Extending OpenStack for cloud-based networking at the edge, in: 2018 IEEE International Conference on Internet of Things, IThings, IEEE, 2018, pp. 162–169.
[39] G. Merlino, D. Bruneo, F. Longo, S. Distefano, A. Puliafito, Cloud-based network virtualization: An IoT use case, in: N. Mitton, M.E. Kantarci, A. Gallais, S. Papavassiliou (Eds.), Ad Hoc Networks, Springer International Publishing, Cham, 2015, pp. 199–210.
[40] G. Tricomi, C. Scaffidi, G. Merlino, F. Longo, S. DIstefano, A. Puliafito, From vertical to horizontal buildings through IoT and software defined approaches, in: Proceedings - 2021 IEEE International Conference on Smart Computing, SMARTCOMP 2021, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 365–370, http://dx.doi.org/10.1109/SMARTCOMP52413.2021.00074, URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85117588551&doi=10.1109%2fSMARTCOMP52413.2021.00074&partnerID=40&md5=4bd6dcfd5d69bbf8770b1232847639fe.
[41] G. Tricomi, C. Scaffidi, G. Merlino, F. Longo, A. Puliafito, S. Distefano, A resilient fire protection system for software-defined factories, IEEE Internet Things J. 10 (4) (2023) 3151–3164, http://dx.doi.org/10.1109/JIOT.2021.3127387, URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85119456832&doi=10.1109%2fJIOT.2021.3127387&partnerID=40&md5=ddd8036f68bad3fc4ed743ebb046ccb6.

Giovanni Merlino is an Associate Professor in Computer Engineering at the Department of Engineering, University of Messina. His research interests currently include IoT for Industry 4.0, Fog computing and the Web of smart things, cyber–physical systems modeled as software-defined infrastructure, smart contracts for auditable access control and delegation, and mobile crowdsensing. He has co-authored over 100 papers, and been involved in several EU projects. He has co-founded an academic spin-off company, smartme.IO, and is holder of a patent. He is IEEE/ACM member.

Giuseppe Tricomi earned his international Ph.D. at the University of Messina in 2021. He is a researcher at the Institute of High-Performance Computing and Networking (ICAR) of the National Research Council of Italy (CNR). His current research interests include Cyber–Physical Systems(CPS), smart environments, Cloud-to-Edge continuum, and cooperative patterns to be applied to these technologies. He is co-authored of 21 papers and has been involved in National and EU projects. He is a member of the team leading the design of Stack4Things, an OpenStack-based Sensing-and-Actuation-as-a-Service framework. He is an IEEE member.

Luca D'Agati is currently a Ph.D. student at the University of Messina, Italy. His main research interests include the Internet of Things (IoT), Cloud computing and their applications in telemedicine, smart cities, and home automation. He has co-authored several papers exploring these technologies, focusing on their integrative potential.

Zakaria Benomar is a PostDoc at the National Institute for Research in Digital Science and Technology (INRIA), Paris, France. He holds a Ph.D. in Cyber–Physical Systems (CPS) from the University of Messina, Italy. His main research interests are currently focused on mobile and distributed systems with particular emphasis on the Internet of Things (IoT), Edge/Fog Computing, and the Web of Things. He is the recipient of the Outstanding Paper Award at the IEEE International Conference on Internet of Things (iThings-2020). He has co-authored over 20 papers in international journals and conferences. He is a member of the team leading the design of Stack4Things, an OpenStack-based Sensing-and-Actuation-as-a-Service framework.

Francesco Longo is Associate Professor at University of Messina. His main research interests include performance and availability evaluation of distributed systems with specific focus on non-Markovian modeling, Internet of Things, and its integration within Infrastructure-as-a-Service Clouds. Most of his work has been published in over 130 papers. He has co-founded an academic spin-off, smartme.IO. He is IEEE member.

Antonio Puliafito is Full Professor in Computer Engineering at University of Messina. His interests include distributed systems, networking, IoT and Cloud computing. He is member of the management board of the National Center of Informatics in Italy (CINI) and the director of the CINI Italian Lab on "Smart Cities & Communities". He is author and co-author of more than 400 scientific papers. He has co-founded an academic spin-off, smartme.IO. He is IEEE member.